# CS 305 Project One Template

## Document Revision History

| **Version** | **Date** | **Author** | **Comments** |
| --- | --- | --- | --- |
| **1.0** | **01/22/25** | **Andrew P. Racic** | **Initial Release** |

## Client



## Instructions

Submit this completed vulnerability assessment report. Replace the bracketed text with the relevant information. In this report, identify your security vulnerability findings and recommend the next steps to remedy the issues you have found.

* Respond to the five steps outlined below and include your findings.
* Respond using your own words. You may also include images or supporting materials. If you include them, make certain to insert them in the relevant locations in the document.
* Refer to the Project One Guidelines and Rubric for more detailed instructions about each section of the template.

**Developer: Andrew P. Racic**

**1. Interpreting Client Needs**

**Client Overview**

Artemis Financial specializes in developing individualized financial plans for customers, including savings, retirement, investments, and insurance. As the company seeks to modernize its operations, enhancing the security of its RESTful web API is crucial to protect against external threats.

**Value of Secure Communications**

Secure communications are vital for Artemis Financial due to the sensitive nature of financial data handled. Protecting client information is essential to maintain trust, ensure compliance with financial regulations, and safeguard against data breaches that could result in significant financial and reputational damage.

**International Transactions**

Although not explicitly mentioned, it's common for financial firms like Artemis Financial to have clients or operations spanning multiple countries. If engaged in international transactions, the company must consider additional security measures and comply with international data protection laws such as the General Data Protection Regulation (GDPR).

**Governmental Restrictions on Secure Communications**

Artemis Financial must adhere to various government regulations dictating how financial data is managed and protected:

* Gramm-Leach-Bliley Act (GLBA): Requires financial institutions to explain their information-sharing practices and safeguard sensitive data.
* Sarbanes-Oxley Act (SOX): Mandates accurate financial reporting and data integrity.
* Payment Card Industry Data Security Standard (PCI DSS): Applies when handling credit card transactions, requiring stringent data security measures.
* State Regulations: Compliance with state-specific laws like the California Consumer Privacy Act (CCPA) may be necessary.

Understanding and complying with these restrictions is essential to avoid legal penalties and ensure the security of client data.

**External Threats Present Now and in the Immediate Future**

**Potential external threats include:**

* Cyber Attacks: Phishing, malware, ransomware, and Distributed Denial of Service (DDoS) attacks targeting financial data.
* API Exploitation: Unsecured APIs may allow unauthorized access to data.
* Data Breaches: Unauthorized access to sensitive client information due to application vulnerabilities.
* Third-Party Risks: Vulnerabilities introduced via open-source libraries or third-party services.
* Advanced Persistent Threats (APTs): Sophisticated, targeted attacks aiming for prolonged network access.

**Modernization Requirements**

* **Role of Open-Source Libraries:**

Open-source libraries can accelerate development but may introduce vulnerabilities if not properly managed.

* + Regular Updates: Ensure all libraries are up-to-date with the latest security patches.
  + Vulnerability Scanning: Implement tools to detect known vulnerabilities in dependencies.
  + License Compliance: Be mindful of licenses associated with open-source components.
* **Evolving Web Application Technologies:**

Adoption of modern frameworks offers improved performance and security features.

* + **Secure Coding Standards:** Implement guidelines to prevent common vulnerabilities.
  + **DevSecOps Integration:** Incorporate security practices into the development lifecycle.
  + **Continuous Monitoring:** Utilize tools for real-time threat detection and response.

**2. Areas of Security**

Based on the vulnerability assessment process flow diagram, the following areas of security apply to Artemis Financial's software application:

1. **Preparation Phase**
   * Relevance: Establishes the scope and objectives of the assessment.
   * Justification: Defines assets at risk, sets boundaries, and ensures compliance with legal and organizational policies.
2. **Discovery Phase**
   * Asset Identification:
     + Relevance: Catalogs all components of the web application, including servers, databases, APIs, and third-party integrations.
     + Justification: Knowing all assets is crucial to identify where vulnerabilities may exist.
   * Vulnerability Identification:
     + Relevance: Uses tools and techniques to discover potential security weaknesses.
     + Justification: Proactively identifies issues before attackers can exploit them.
3. **Analysis Phase**
   * Vulnerability Analysis:
     + Relevance: Examines identified vulnerabilities to determine their impact and exploitability.
     + Justification: Prioritizes vulnerabilities based on risk levels, allowing focused remediation efforts.
   * **Risk Assessment:**
     + Relevance: Evaluates the likelihood of vulnerabilities being exploited and potential consequences.
     + Justification: Assists in deciding which vulnerabilities to address first based on their criticality.
4. **Remediation Phase**
   * **Mitigation Planning:**
     + Relevance: Develops strategies to address identified vulnerabilities.
     + Justification: Provides a roadmap for remediation efforts aligned with organizational priorities and resources.
   * **Implementation:**
     + Relevance: Applies fixes, updates, or changes to eliminate or reduce vulnerabilities.
     + Justification: Improves the application's security posture by rectifying weaknesses.
5. **Validation Phase**
   * **Verification Testing:**
     + Relevance: Tests the application post-remediation to ensure vulnerabilities have been effectively addressed.
     + Justification: Confirms that mitigation efforts are successful and haven't introduced new issues.
   * **Documentation:**
     + Relevance: Records findings, actions taken, and outcomes of the assessment.
     + Justification: Provides an audit trail and reference for future assessments and compliance purposes.
6. **Maintenance Phase**
   * **Continuous Monitoring:**
     + Relevance: Ongoing surveillance of the application for new vulnerabilities.
     + Justification: Ensures sustained security by promptly identifying and addressing emerging threats.
   * **Updates and Patch Management:**
     + Relevance: Regularly applies updates to software and dependencies.
     + Justification: Keeps the application protected against known vulnerabilities discovered over time.
7. **Compliance and Policy Enforcement**
   * Relevance: Ensures the application adheres to relevant laws, regulations, and organizational security policies.
   * Justification: Avoids legal penalties and maintains the company's reputation by meeting compliance requirements.

**Justification for Relevance to Artemis Financial**

* Financial Data Sensitivity: Handling sensitive financial information necessitates thorough assessment across all phases to protect against data breaches.
* Regulatory Compliance: Compliance and policy enforcement are critical due to the heavily regulated financial industry.
* Evolving Threat Landscape: Continuous monitoring and maintenance address the dynamic nature of cybersecurity threats, ensuring protection against new vulnerabilities.
* Integration of Third-Party Components: Discovery and analysis help identify vulnerabilities in third-party libraries and APIs common in modern web applications.
* Organizational Preparedness: Preparation and remediation ensure readiness to respond effectively to security incidents, minimizing potential damage.

**3. Manual Review**

A manual inspection of the codebase revealed the following vulnerabilities:

1. **Hardcoded Database Credentials**
   * Location: DocData.java
   * Issue: Database username and password are hardcoded.

java

Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "root");

* + Risk: Exposes sensitive credentials, allowing unauthorized database access if the code is compromised.
  + Recommendation: Remove hardcoded credentials. Use environment variables or a secure configuration management system to store and retrieve database credentials.

1. **Lack of Input Validation and Sanitization**
   * Location: CRUDController.java and GreetingController.java

java

public CRUD CRUD(@RequestParam(value="business\_name") String name) { ... }

public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) { ... }

* + Risk: Vulnerable to injection attacks (e.g., SQL Injection, XSS), leading to unauthorized actions or data breaches.
  + Recommendation: Implement robust input validation and sanitization. Validate inputs against expected patterns and sanitize them to remove malicious content.

1. **Potential SQL Injection Vulnerability**
   * Location: DocData.java

java

public void read\_document(String key, String value) { ... }

* + Risk: Attackers could manipulate inputs to inject malicious SQL commands, compromising the database.
  + Recommendation: Use prepared statements or parameterized queries for all database interactions involving user input.

1. **Improper Exception Handling Revealing Sensitive Information**
   * Location: DocData.java

java

catch (SQLException e) {

e.printStackTrace();

}

* + Risk: Stack traces can expose internal details to attackers.
  + Recommendation: Use a logging framework to log exceptions securely. Provide generic error messages without revealing technical details.

1. **Weak Access Modifiers and Lack of Encapsulation**
   * Location: customer.java

java

int account\_balance;

* + Risk: Other classes can access and modify these fields directly, leading to unauthorized data manipulation.
  + Recommendation: Declare fields as private and provide controlled access through public getter and setter methods with necessary validations.

1. **Missing Authentication and Authorization Controls**
   * Location: Application-wide issue.
   * Risk: Unauthorized users can access sensitive endpoints and data without restrictions.
   * Recommendation: Implement authentication (e.g., JWT, OAuth) to verify user identity. Enforce authorization checks based on user roles and permissions.
2. **Exposure of Sensitive Information via Unrestricted Methods**
   * Location: customer.java

java

public int showInfo() {

// code to show customer information

return this.account\_number;

}

* + Risk: Allows unauthorized retrieval of customer data, leading to privacy breaches.
  + Recommendation: Implement access control checks within methods. Ensure only authorized users can view customer information.

1. **Incomplete Input Validation in Date and Time Handling**
   * Location: myDateTime.java

java

void setMyDateTime(int seconds, int minutes, int hour) { ... }

* + Risk: Accepting invalid time values can cause errors or inconsistent data handling.
  + Recommendation: Validate input parameters to ensure seconds are between 0–59, minutes between 0–59, and hours between 0–23.

1. **Potential Denial of Service via Unrestricted Resource Consumption**
   * Location: GreetingController.java

java

private final AtomicLong counter = new AtomicLong();

* + Risk: An attacker could flood the endpoint, causing resource exhaustion or integer overflow.
  + Recommendation: Implement rate limiting on the endpoint. Monitor and manage resource usage effectively.

1. **Absence of Secure Communication Enforcement**
   * Location: Application-wide issue.
   * Risk: Data transmitted over unsecured channels can be intercepted.
   * Recommendation: Enforce HTTPS throughout the application. Implement security headers like Strict-Transport-Security, Content-Security-Policy, and X-Content-Type-Options.

**Summary**

The manual code inspection uncovered critical vulnerabilities, including hardcoded credentials, lack of input validation, potential for injection attacks, insufficient exception handling, and missing authentication controls. Addressing these vulnerabilities is essential to protect the application from external threats and safeguard sensitive client data.

**Next Steps**

* Prioritize Vulnerabilities: Address critical issues first, such as hardcoded credentials and lack of input validation.
* Implement Security Best Practices: Adopt a comprehensive approach to secure coding, including regular code reviews and security training.
* Enhance Application Architecture: Introduce frameworks and tools that support secure development, authentication, and authorization.
* Continuous Monitoring: Establish processes for ongoing security assessment and vulnerability management.

**4. Static Testing**

Using the OWASP Dependency-Check Maven plug-in, several known vulnerabilities in third-party libraries were identified:

1. Vulnerabilities in Bouncy Castle Provider (bcprov-jdk15on-1.46.jar)
   * Vulnerability IDs: CVE-2016-1000341, CVE-2018-5382 (Total CVEs: 20).
   * Description: Multiple vulnerabilities leading to information disclosure and cryptographic weaknesses.
   * Recommended Solution: Upgrade to bcprov-jdk15on-1.70 or later. Review cryptographic usage to ensure compliance with current security standards.
2. Critical Vulnerabilities in Spring Boot (spring-boot-2.2.4.RELEASE.jar)
   * Vulnerability IDs: CVE-2020-5421, CVE-2022-22965 ("Spring4Shell"), CVE-2021-22096.
   * Description: Allows attackers to execute arbitrary code, escalate privileges, or cause denial of service.
   * Recommended Solution: Upgrade to Spring Boot 2.6.7 or later. Apply patches provided by Spring.
3. Vulnerabilities in Logback (logback-classic-1.2.3.jar, logback-core-1.2.3.jar)
   * Vulnerability IDs: CVE-2021-42550, CVE-2017-5929 (Total CVEs: 6).
   * Description: Attackers can manipulate logging configurations to execute code or disrupt services.
   * Recommended Solution: Upgrade to Logback 1.2.11 or later. Disable JNDI functionality if not required.
4. Critical Vulnerabilities in Log4j API (log4j-api-2.12.1.jar)
   * Vulnerability IDs: CVE-2021-44228 ("Log4Shell"), CVE-2021-45046, CVE-2021-45105, CVE-2021-44832, CVE-2020-9488.
   * Description: Allows remote code execution and denial of service attacks through malicious input exploiting the logging mechanism.
   * Recommended Solution: Upgrade to Log4j 2.17.1 or later. Remove or disable the JNDI lookup feature immediately.
5. Multiple Vulnerabilities in SnakeYAML (snakeyaml-1.25.jar)
   * Vulnerability IDs: CVE-2017-18640, CVE-2022-25857 (Total CVEs: 10).
   * Description: Vulnerable to attacks consuming excessive resources or executing arbitrary code via crafted YAML content.
   * Recommended Solution: Upgrade to SnakeYAML 1.30 or later. Configure safe loading mechanisms.
6. Vulnerabilities in Jackson Databind (jackson-databind-2.10.2.jar)
   * Vulnerability IDs: CVE-2019-12384, CVE-2020-36518 (Total CVEs: 6).
   * Description: Unsafe deserialization leading to potential remote code execution.
   * Recommended Solution: Upgrade to Jackson Databind 2.13.3 or later. Disable default typing or strictly define allowable types.
7. Critical Vulnerabilities in Apache Tomcat (tomcat-embed-core-9.0.30.jar)
   * Vulnerability IDs: CVE-2020-1938 ("Ghostcat"), CVE-2021-25329, CVE-2022-23181 (Total CVEs: 27).
   * Description: Vulnerabilities leading to information disclosure, session hijacking, and remote code execution.
   * Recommended Solution: Upgrade to Apache Tomcat 9.0.62 or later. Disable the AJP connector if not in use.
8. Vulnerabilities in Hibernate Validator (hibernate-validator-6.0.18.Final.jar)
   * Vulnerability IDs: CVE-2019-10219, CVE-2020-10693.
   * Description: Potential for remote code execution or injection of malicious scripts.
   * Recommended Solution: Upgrade to Hibernate Validator 6.1.7.Final or later. Sanitize constraint messages and avoid including untrusted data.
9. Multiple Vulnerabilities in Spring Framework Dependencies
   * Affected Dependencies: spring-web-5.2.3.RELEASE.jar, spring-beans-5.2.3.RELEASE.jar, spring-webmvc-5.2.3.RELEASE.jar, spring-context-5.2.3.RELEASE.jar, spring-expression-5.2.3.RELEASE.jar.
   * Vulnerability IDs: CVE-2020-5398, CVE-2021-22118, CVE-2022-22965 ("Spring4Shell"), CVE-2022-22950.
   * Description: May allow attackers to execute code remotely, cause denial of service, or access sensitive information.
   * Recommended Solution: Upgrade to Spring Framework 5.3.18 or later. Apply additional security patches.

**General Recommendations from Static Testing**

* **Update All Vulnerable Dependencies:**

Regularly check for updates and patches. Use tools like Maven's versions-maven-plugin to manage dependencies.

* **Implement Dependency Management Best Practices:**
  + Avoid Vulnerable Transitive Dependencies: Exclude or override versions that bring in vulnerabilities.

xml

<dependency>

<groupId>example</groupId>

<artifactId>example-dependency</artifactId>

<version>1.0.0</version>

<exclusions>

<exclusion>

<groupId>vulnerable</groupId>

<artifactId>vulnerable-lib</artifactId>

</exclusion>

</exclusions>

</dependency>

* + Use Dependency Convergence: Ensure consistent dependency versions to prevent unexpected behaviors.
* **Automate Security Scanning:**

Integrate dependency checks into the CI/CD pipeline. Use tools like OWASP Dependency-Check, Snyk, or Whitesource Bolt for ongoing monitoring.

* **Establish a Vulnerability Management Process:**

Assign responsibility for tracking and addressing vulnerabilities. Prioritize remediation efforts based on severity and potential impact.

**5. Mitigation Plan**

**Addressing Manual Review Vulnerabilities**

1. **Hardcoded Database Credentials**
   * Action: Refactor DocData.java to remove hardcoded credentials. Use external configuration files or environment variables to store sensitive information securely. Utilize a secrets management service or encrypted configuration management.
2. **Lack of Input Validation and Sanitization**
   * Action: Implement input validation using annotations like @Valid, @NotNull, @Size. Sanitize inputs using libraries like OWASP Java Encoder. Update CRUDController.java and GreetingController.java to include validation logic.
3. **Potential SQL Injection Vulnerability**
   * Action: Refactor database interaction code in DocData.java to use prepared statements and validate inputs before execution.

java

String query = "SELECT \* FROM documents WHERE key = ?";

PreparedStatement pstmt = con.prepareStatement(query);

pstmt.setString(1, key);

ResultSet rs = pstmt.executeQuery();

1. **Improper Exception Handling Revealing Sensitive Information**
   * Action: Replace printStackTrace() with proper logging calls using a logging framework. Ensure error messages do not reveal sensitive system information.
2. **Weak Access Modifiers and Lack of Encapsulation**
   * Action: Modify customer.java to encapsulate fields by declaring them as private. Provide controlled access through getter and setter methods with necessary validations.

java

public class Customer {

private int accountNumber;

private int accountBalance;

public int getAccountBalance() {

return accountBalance;

}

public void deposit(int amount) {

if (amount > 0) {

accountBalance += amount;

}

}

}

1. **Missing Authentication and Authorization Controls**
   * Action: Set up Spring Security in the application. Implement authentication mechanisms and define user roles and permissions. Secure controllers using annotations like @PreAuthorize.
2. **Exposure of Sensitive Information via Unrestricted Methods**
   * Action: Update methods in customer.java to include authorization checks. Verify user permissions before returning sensitive information and log access for auditing.
3. **Incomplete Input Validation in Date and Time Handling**
   * Action: Implement validation logic in myDateTime.java to ensure time values are within valid ranges.

java

public void setMyDateTime(int seconds, int minutes, int hour) {

if (seconds >= 0 && seconds < 60 &&

minutes >= 0 && minutes < 60 &&

hour >= 0 && hour < 24) {

this.mySecond = seconds;

this.myMinute = minutes;

this.myHour = hour;

} else {

throw new IllegalArgumentException("Invalid time values provided.");

}

}

1. **Potential Denial of Service via Unrestricted Resource Consumption**
   * Action: Introduce rate limiting and safeguard the counter logic in GreetingController.java. Handle counter overflow scenarios.
2. **Absence of Secure Communication Enforcement**
   * Action: Update application configurations to enforce HTTPS. Implement security headers such as Strict-Transport-Security, Content-Security-Policy, X-Content-Type-Options, X-Frame-Options, and X-XSS-Protection. Ensure SSL/TLS certificates are properly installed and maintained.

**Addressing Static Testing Vulnerabilities**

1. **Outdated and Vulnerable Dependencies**
   * Action: Update all vulnerable dependencies to their latest secure versions:
     + Spring Boot: Upgrade to 2.6.x.
     + Spring Framework Components: Upgrade to versions compatible with the updated Spring Boot.
     + Log4j API: Upgrade to 2.17.1 or later.
     + Logback: Upgrade to 1.2.11 or later.
     + Jackson Databind: Upgrade to 2.13.3 or the latest stable version.
     + SnakeYAML: Upgrade to 1.30 or later.
     + Apache Tomcat Embed Core: Upgrade to 9.0.62 or later.
     + Bouncy Castle: Upgrade to bcprov-jdk15on 1.70 or later.
   * Test Compatibility: Thoroughly test the application after upgrades to ensure functionality.
   * Use Dependency Management Tools: Leverage Maven's dependency management to control versions and prevent conflicts.
2. **Implement Continuous Dependency Monitoring**
   * Action: Integrate tools like OWASP Dependency-Check into the build process. Use tools like Renovate or Dependabot for automatic updates. Stay informed through security mailing lists.
3. **Address Transitive Dependencies**
   * Action: Use Maven's dependency:tree to identify dependencies. Exclude or override vulnerable transitive dependencies.

**General Security Enhancements**

1. **Implement Secure Coding Practices**
   * Training and Guidelines: Provide developers with training on OWASP Top Ten vulnerabilities and secure coding standards. Establish coding guidelines emphasizing security throughout the development lifecycle.
   * Code Reviews: Incorporate security-focused code reviews.
   * Action: Develop a secure coding policy and ensure team adherence.
2. **Enhance Logging and Monitoring**
   * Implement Centralized Logging: Use a centralized logging system to collect and analyze logs securely.
   * Monitor for Suspicious Activities: Set up alerts for unusual patterns or potential incidents.
   * Protect Logs: Ensure logs do not contain sensitive information and are stored securely with access controls.
   * Action: Deploy a robust logging and monitoring system with security considerations.
3. **Establish an Incident Response Plan**
   * Preparation: Develop a documented plan outlining steps for security breaches.
   * Roles and Responsibilities: Assign team roles for incident detection, analysis, containment, eradication, and recovery.
   * Regular Drills: Conduct drills to test the plan's effectiveness.
   * Action: Create and maintain an incident response plan.
4. **Perform Regular Security Assessments**
   * Penetration Testing: Schedule periodic tests to identify vulnerabilities not apparent through code review.
   * Vulnerability Scanning: Use automated tools to scan applications and infrastructure.
   * Action: Incorporate regular assessments into development and deployment cycles.
5. **Compliance and Regulatory Adherence**
   * Understand Regulatory Requirements: Ensure compliance with regulations like GDPR, GLBA, and PCI DSS.
   * Document Policies and Procedures: Maintain updated security policies, procedures, and compliance records.
   * Action: Review and update compliance measures, conduct audits.
6. **Secure Configuration Management**
   * Configuration Baselines: Establish secure baselines for servers, databases, and devices.
   * Version Control: Use version control for configuration files.
   * Limit Exposure: Close unnecessary ports, disable unused services.
   * Action: Implement secure configuration management practices.

**Conclusion**

By addressing the identified vulnerabilities through this comprehensive mitigation plan, Artemis Financial can significantly enhance the security of its software application. Implementing these steps will:

* Protect sensitive client and company data.
* Prevent unauthorized access and potential breaches.
* Ensure compliance with industry regulations and standards.
* Foster a culture of security awareness within the development team.

**Next Steps**

* Prioritize Actions: Begin with critical vulnerabilities posing the highest risk.
* Allocate Resources: Assign responsibilities to team members for each action item.
* Set Timelines: Establish clear deadlines for implementing mitigation steps.
* Validate Fixes: After applying mitigations, thoroughly test the application to ensure vulnerabilities are resolved.
* Monitor and Update: Continuously monitor the application's security posture and update practices as needed.

By proactively addressing these vulnerabilities and implementing robust security practices, Artemis Financial can enhance its security posture, comply with regulatory requirements, and maintain the trust of its clients.